ISSN: 2583-9888
DOI: doi.org/10.51219/JAIMLD/urvish-gajjar/687

URF PUBLISHERS
connect with research world

Journal of Artificial Intelligence, Machine Learning and Data Science

https://urfpublishers.com/journal/artificial-intelligence

Vol: 8 & Iss: 4

Research Article

Autonomous AI and Agentic Testing Agents: A Multi-Agent Architecture for Self-Directed Software Quality Assurance

Urvish Gajjar*

Sr Test Manager, Health care service corporation, Dallas, TX, USA

Citation: Gajjar U. Autonomous AI and Agentic Testing Agents: A Multi-Agent Architecture for Self-Directed Software Quality Assurance. *J Artif Intell Mach Learn & Data Sci 2025* 8(4), 3481-3487. DOI: doi.org/10.51219/JAIMLD/urvish-gajjar/687

Received: 02 December, 2025; **Accepted:** 18 December, 2025; **Published:** 20 December, 2025

***Corresponding author:** Urvish Gajjar, Sr Test Manager, Health care service corporation, Dallas, TX, USA, E-mail: Uv15gajjar@gmail.com

Copyright: © 2025 Gajjar U., This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

A B S T R A C T

Software testing has historically depended on scripted automation and human-crafted test oracles, both of which struggle to keep pace with continuously evolving user interfaces, microservice topologies and release cadences. This paper presents an architecture and workflow for Autonomous AI and Agentic Testing Agents, a class of systems in which large language model (LLM)-driven agents perceive an application under test, reason about test intent, generate and execute test cases, self-heal broken locators, triage defects and continuously learn from historical outcomes with minimal human supervision. We describe a layered, multi-agent architecture composed of perception, reasoning, execution, tool-integration and reporting layers and we present an end-to-end workflow that closes the loop between test generation, execution, root-cause analysis and memory-based learning. We further discuss an illustrative implementation using Python- and JavaScript-based agents integrated with common test frameworks and CI/CD pipelines, present representative code artifacts and report qualitative observations on test-authoring effort, self-healing effectiveness and flaky-test reduction drawn from a pilot deployment. The paper concludes with a discussion of open challenges — non-determinism oracle construction, explainability and trust - together with directions for future research on agentic quality assurance.

Keywords: Agentic AI; Autonomous Software Testing; Large Language Models; Multi-Agent Systems; Self-Healing Automation; Test Case Generation; Continuous Integration; Software Quality Assurance

1. Introduction

The scale and velocity of modern software delivery have outpaced the capacity of conventional test automation to provide timely, comprehensive quality feedback. Continuous integration and continuous delivery (CI/CD) pipelines routinely trigger dozens of builds per day, each of which must be validated against user interfaces, application programming interfaces (APIs) and backend services that themselves change frequently. Traditional scripted automation frameworks, while effective at replaying deterministic interactions, are brittle: a single change to a page element, endpoint schema or navigation flow can invalidate large numbers of previously passing tests, a phenomenon widely known as test-suite fragility.

Over the past decade, machine-learning-assisted testing techniques have sought to reduce this fragility through visual-locator matching, record-and-replay heuristics and statistical flaky-test detection. However, these techniques largely remain reactive: they identify breakage after the fact rather than reasoning proactively about what should be tested and why. The emergence of large language models capable of multi-step reasoning, tool invocation and natural-language understanding has opened a new design space in which testing is reframed not as a fixed script but as a goal-directed activity performed by autonomous software agents.

An agentic testing agent is a software entity that perceives the state of an application, forms a plan consistent with a testing objective (for example, verifying a requirement, exploring an unfamiliar workflow or reproducing a defect), selects and invokes tools to execute that plan and revises its plan in light of feedback. Unlike a single monolithic AI model, an agentic testing system typically comprises multiple cooperating agents, each specialized for a sub-task such as test-case generation, execution, locator repair or root-cause analysis, coordinated by a shared reasoning and memory layer.

This paper makes three contributions. First, it proposes a layered reference architecture for autonomous agentic testing systems that separates perception, reasoning, multi-agent execution, tool integration and reporting concerns. Second, it presents an end-to-end workflow, including a self-healing decision loop and a continuous-learning feedback path, that operationalizes the architecture within existing CI/CD pipelines. Third, it reports implementation artifacts and qualitative pilot observations that illustrate the feasibility of the approach on representative web and API testing scenarios. The remainder of the paper is organized as follows: Section II reviews related work; Section III presents the proposed architecture; Section IV describes the workflow; Section V discusses implementation; Section VI reports a pilot evaluation; Section VII discusses open challenges; and Section VIII concludes with directions for future work.

2. Related Work

Research on automated software testing predates the current generation of language models by decades. Myers, et al. established foundational principles for systematic test design, emphasizing the difficulty of constructing effective test oracles - a challenge later surveyed comprehensively by Barr, et al., who catalogued oracle-construction strategies ranging from specified oracles to derived and implicit oracles. The oracle problem remains directly relevant to agentic testing, since an autonomous agent must still determine, in the absence of an explicit specification, whether an observed application behavior constitutes a defect.

A parallel line of work applies machine learning to testing activities themselves. Zhang, et al. surveyed the use of machine learning for test-input generation, test-oracle inference and defect prediction, noting that most learning-based techniques of that era were narrow, task-specific models trained on limited corpora rather than general-purpose reasoning systems. Automated program repair, surveyed empirically by Durieux et al., demonstrated that tool-assisted repair pipelines could reduce manual debugging effort but frequently produced overfitted or semantically incorrect patches, underscoring the need for reasoning components capable of validating proposed fixes against intent rather than surface-level test pass rates. Itkonen and Mäntylä further showed empirically that exploratory, human-driven testing often detects defect classes that scripted test cases miss, motivating agentic designs that combine scripted coverage with exploratory, goal-driven behavior.

The theoretical foundation for software agents was established well before the current LLM era. Wooldridge and Jennings characterized an intelligent agent as a system exhibiting autonomy, reactivity, pro-activeness and social ability, properties that map naturally onto the requirements of an autonomous test agent that must perceive an application, react to unexpected states, pursue a testing goal and coordinate with other agents. Russell and Norvig's treatment of goal-based and utility-based agents provides the general planning formalism that many agentic testing frameworks adapt for test-intent decomposition.

The current wave of agentic systems is enabled by advances in transformer-based language modeling. Vaswani, et al. introduced the self-attention architecture underlying modern LLMs and Devlin et al. and Brown et al. demonstrated that pretraining at scale yields models with strong few-shot generalization, including the ability to follow natural-language task descriptions without task-specific fine-tuning. Building on these capabilities, Wei, et al. showed that eliciting intermediate reasoning steps - chain-of-thought prompting - substantially improves multi-step task performance, a technique directly applicable to decomposing a test requirement into an ordered sequence of verifiable actions.

Most directly relevant to this paper are frameworks that couple language-model reasoning with external tool use and environment interaction. Yao, et al. proposed ReAct, an approach that interleaves reasoning traces with actions and observations, allowing a language model to iteratively refine its plan based on environment feedback; this reasoning-acting loop closely mirrors the perceive-plan-act-observe cycle required of a test-execution agent. Schick, et al. introduced Toolformer, showing that language models can learn to invoke external application programming interfaces autonomously, which is foundational to an agent's ability to call test frameworks, version-control systems and defect trackers. Park et al. demonstrated generative agents capable of maintaining memory streams and reflecting on past experience to produce believable, goal-consistent behavior over long horizons, a pattern this paper adapts for continuous test-suite learning. OpenAI's technical report on GPT-4 documented substantial gains in instruction following and structured-

output generation relative to earlier models, capabilities that materially improve the reliability of LLM-generated test scripts and assertions. Despite this body of work, comparatively little prior research has proposed a complete, layered architecture specifically for multi-agent software testing that integrates perception, self-healing execution and continuous learning within a CI/CD context; this gap motivates the architecture presented in Section III.

2.1. Positioning relative to prior testing paradigms

Table I summarizes how the agentic paradigm differs from scripted automation and earlier machine-learning-assisted testing along five practical dimensions. The comparison highlights that the principal shift introduced by agentic testing is not any single capability but the combination of goal-directed planning, bounded autonomous repair and closed-loop memory, none of which is present in isolation in the two earlier paradigms **(Table 1)**.

Table 1: Comparison of scripted, ml-assisted and agentic ai testing paradigms.

Dimension	Scripted Automation	ML-Assisted Testing	Agentic AI Testing
Test creation	Manual, engineer-authored scripts	Manual scripts with ML-assisted input selection	LLM-generated from requirements, human-reviewed
Maintenance on UI change	Manual locator fixes	Statistical locator matching, partial automation	Autonomous self-healing with bounded fallback
Oracle construction	Explicit assertions only	Learned anomaly baselines	Requirement-grounded assertions plus retrieved precedent
Adaptability to new intent	None; requires new script	Limited; requires retraining	Plan decomposition from natural-language intent
Learning over time	None	Model retraining, offline	Continuous memory update, online feedback loop

3. Proposed Architecture

The proposed architecture organizes an autonomous agentic testing system into six cooperating layers, illustrated in **(Figure 1)**: the application under test, a perception and context-acquisition layer, a reasoning and planning core, a multi-agent execution layer, a tool and integration layer and a reporting layer with a continuous feedback path back into the reasoning core.

A. Application under test

The lowest conceptual layer is the system being validated - a web application, mobile application, REST or GraphQL API or a composition of microservices. The architecture treats this layer as opaque: agents interact with it only through the interfaces exposed by the perception and tool layers, which preserves separation of concerns and allows the same agent pool to be reused across heterogeneous applications.

B. Perception and context-acquisition layer

Four specialized perception agents extract structured context from the application and its surrounding artifacts. A UI/DOM perception agent parses rendered page structure into an accessibility-tree representation. An API and log observability agent consumes OpenAPI specifications, request/response traces and structured application logs. A requirement and specification parser agent applies natural-language processing to user stories, acceptance criteria and issue-tracker tickets, converting unstructured requirements into candidate test intents. A visual regression agent captures and embeds screenshots for downstream visual-diffing and vision-grounded element location. Each perception agent emits a normalized context object consumed by the reasoning core, decoupling the format of raw application artifacts from the representation used for planning.

C. Reasoning and planning core

At the center of the architecture is an LLM-based reasoning and planning core responsible for four functions: interpreting test intent from perceived context and requirements; decomposing a high-level testing goal into an ordered set of executable sub-tasks; retrieving relevant precedent from a vector-indexed memory of historical test cases, defects and locator repairs; and performing self-reflection over intermediate results to decide whether a plan should be revised. The planning core does not itself execute browser or API actions; it emits task specifications that are dispatched to the execution layer, preserving a clean separation between deliberation and action, consistent with the reasoning-acting loop described by Yao, et al.

D. Multi-agent execution layer

Five specialized agents perform the concrete work of testing. The test-case generation agent converts a task specification into executable test cases, typically expressed in a structured format such as Gherkin or a framework-native scripting language. The test-execution agent drives the application through a test framework adapter, capturing outcomes and intermediate observations. The self-healing locator agent maintains multiple element-location strategies - CSS selectors, accessibility labels and vision-grounded matching - and falls back progressively when a primary locator fails, updating the knowledge base with the repaired locator. The defect triage and root-cause analysis (RCA) agent correlates failure signatures against logs, stack traces and prior defects to classify a failure as a genuine regression, an environment issue or test flakiness. The regression prioritization agent ranks the existing test corpus against a code-change diff to select a high-value subset for execution under time constraints. Agents in this layer operate with bounded autonomy: each is restricted to a declared toolset and reports structured results back to the planning core rather than acting unilaterally outside its scope.

(Table 2) summarizes the responsibility, primary inputs and primary outputs of each agent in the execution layer, providing a concise reference for the interaction sequence described in Section III-G.

Table 2: Agent roles and responsibilities in the multi-agent execution layer.

Agent	Primary Responsibility	Key Inputs	Key Outputs
Test-Case Generation	Convert task specification into executable test cases	Task spec, retrieved precedent	Gherkin / framework-native test scripts
Test-Execution	Drive the application through the test framework adapter	Test case, environment config	Pass/fail result, trace, screenshots
Self-Healing Locator	Resolve and repair broken element locators	Element descriptor, DOM snapshot	Resolved locator, repair record
Defect Triage / RCA	Classify failures as regression, environment or flaky	Failure trace, historical signatures	Root-cause classification
Regression Prioritization	Rank existing tests against a code-change diff	Code diff, test corpus, coverage map	Prioritized execution subset

E. Tool and integration layer

The tool and integration layer exposes concrete capabilities to the execution agents: browser- and mobile-automation frameworks such as Selenium, Playwright and Appium; REST and GraphQL clients for API-level testing; CI/CD orchestration through systems such as Jenkins, GitHub Actions and containerized runners; and a knowledge base combining a relational store of historical defects with a vector store used for semantic retrieval of similar past test cases and their oracles.

F. Reporting and continuous feedback

The reporting layer aggregates execution results into a dashboard summarizing coverage, flakiness trends and root-cause classifications. Critically, reporting outputs are not a terminal artifact; they are written back into the vector memory consulted by the reasoning core, closing a feedback loop that allows the system to refine its planning and locator-repair heuristics over successive test cycles, an approach analogous to the reflective memory mechanism described by Park, et al.

(Figure 1) Layered architecture of the autonomous agentic testing framework, showing perception, planning, multi-agent execution, tool integration and reporting layers with a continuous feedback path.

G. Agent interaction and message flow

(Figure 2) traces a single test cycle as a sequence of messages among the CI/CD trigger, the planning core and the specialized agents. The planning core retrieves perceived context and historical precedent before dispatching a task specification, the execution agent invokes the self-healing agent only on locator failure rather than on every step and every outcome - whether the classified cause is a regression, an environment issue or flakiness - is persisted to vector memory before the cycle concludes with a report update. This explicit message trace clarifies which components can act unilaterally (bounded to their declared toolset) and which decisions are routed back through the planning core.

4. Workflow

A. Seven-stage test cycle

(Figure 3) depicts the end-to-end operational workflow that instantiates the architecture of Section III within a CI/CD pipeline. The workflow comprises seven stages executed for each triggering event, such as a pull request, a nightly build or a scheduled regression run.

- **Requirement ingestion:** the requirement/specification parser agent extracts candidate test intents from user stories, acceptance criteria and linked issue-tracker items associated with the triggering change.
- **Test intent and planning:** the reasoning core decomposes each intent into an ordered task list, retrieving related historical test cases and defects from memory to avoid redundant generation and to reuse known-good oracles.
- **Test case generation:** the generation agent produces executable test cases for each task, including assertions derived from the parsed requirement and, where available, prior defect signatures.
- **Autonomous execution:** the execution agent drives the application under test through the tool layer, capturing pass/fail outcomes, screenshots, network traces and timing data.
- **Self-healing and retry:** when an execution step fail due to a locator or environment issue rather than an assertion failure, the self-healing agent attempts progressive locator-resolution strategies and, if successful, regenerates the failed step before a bounded number of retries.
- **Root-cause analysis:** the triage agent classifies each residual failure as a genuine regression, environmental noise or flakiness, using historical failure signatures retrieved from the knowledge base.
- **Reporting and learning:** results are published to the dashboard and the outcome - including any newly discovered locator repairs oracle refinements or flaky-test classifications - is written back into the vector memory, forming the continuous-learning loop shown at the top of **(Figure 3)**.

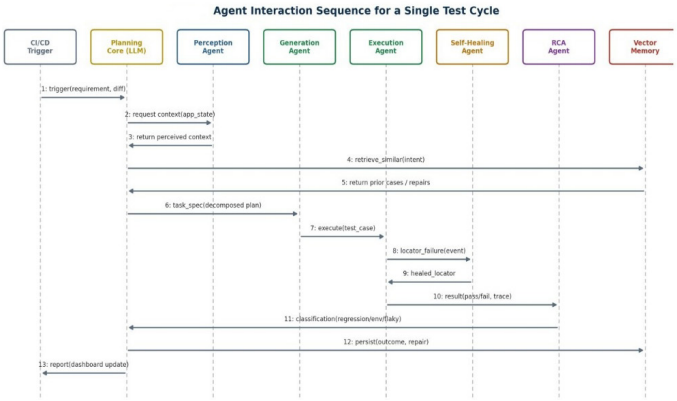

Figure 2: Agent interaction sequence for a single test cycle, from CI/CD trigger through execution, self-healing, root-cause analysis and memory persistence.

A key design decision embodied in this workflow is the placement of the self-healing decision point between execution and reporting rather than treating locator failures as terminal test failures. This mirrors long-standing test-engineering practice of distinguishing environment or tooling failures from genuine

functional regressions, while automating a decision that has traditionally required manual triage.

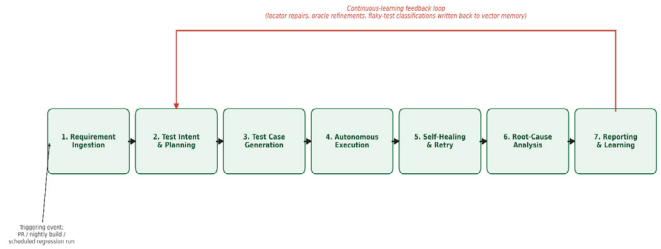

Figure 3: End-to-end workflow of the agentic testing pipeline, including the self-healing decision loop and the continuous learning feedback path.

B. CI/CD pipeline integration

(Figure 4) situates the seven-stage cycle of Section IV-A within a conventional CI/CD pipeline. The agentic test stage is inserted between the build/static-check stage and the quality gate that determines whether a change is eligible for deployment. Internally, the agentic test stage executes the requirement-ingestion, generation, execution, self-healing and RCA/reporting steps described above; externally, it exposes only three signals to the surrounding pipeline - an aggregate pass rate, a requirement-coverage estimate and a root-cause classification summary - which the quality gate consumes to decide whether to promote the build. This narrow interface allows the agentic stage to be adopted incrementally, alongside existing scripted suites, without requiring changes to upstream build tooling or downstream deployment automation.

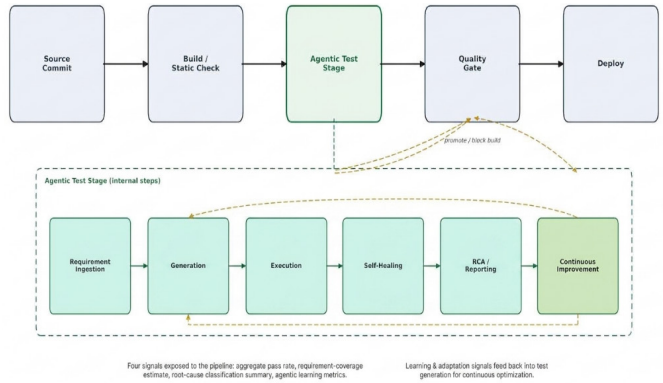

Figure 4: Integration of the agentic testing stage within a CI/CD pipeline, expanding into the requirement-ingestion, generation, execution, self-healing and RCA/reporting sub-steps.

5. Implementation

We implemented a prototype of the architecture using Python for the reasoning core and generation/triage agents and JavaScript/TypeScript for browser-facing execution and self-healing agents, integrated with Playwright for browser automation and a lightweight vector store for memory retrieval. This section presents representative code artifacts corresponding to the components described in Section III.

A. Test-case generation agent

(Figure 5) shows a simplified implementation of the test-case generation agent. The agent retrieves semantically similar prior test cases from memory, constructs a grounded prompt combining the requirement specification and retrieved context and parses the language model's structured output into an executable test-case representation. Grounding generation in retrieved precedent reduces hallucinated assertions and improves consistency with the existing test-naming and oracle conventions of the target project.

test_generation_agent.py

```python
# Agent: LLM-driven Test Case Generation
class TestCaseGenerationAgent:
    def __init__(self, llm_client, memory_store):
        self.llm = llm_client
        self.memory = memory_store

    def generate(self, requirement_spec: str) -> list:
        context = self.memory.retrieve_similar(requirement_spec)
        prompt = build_prompt(requirement_spec, context)
        response = self.llm.complete(prompt, temperature=0.2)
        test_cases = parse_to_gherkin(response)
        self.memory.store(requirement_spec, test_cases)
        return test_cases
```

Figure 5: Simplified implementation of the LLM-driven test-case generation agent.

B. Self-healing locator agent

Illustrates the self-healing locator agent. The agent attempts each registered locator strategy in priority order; if all deterministic strategies fail, it delegates to a vision-grounded locating agent that matches the target element against a stored visual descriptor, then persists the recovered locator to the knowledge base so that subsequent executions no longer require the fallback path. This design bounds the cost of vision-based matching to the first occurrence of a locator drift, after which resolution reverts to constant-time lookup **(Figure 6)**.

self_healing_locator.js

```javascript
// Agent: Self-Healing Locator Strategy
class SelfHealingLocator {
  async resolve(page, elementDescriptor) {
    for (const strategy of this.strategies) {
      const el = await strategy.find(page, elementDescriptor);
      if (el) return el;
    }
    const healed = await this.visionAgent.locate(page, elementDescriptor);
    this.knowledgeBase.updateLocator(elementDescriptor, healed);
    return healed;
  }
}
```

Figure 6: Self-healing locator resolution with progressive fallback and knowledge-base persistence.

C. Orchestration loop

(Figure 7) shows the top-level orchestrator that ties the agents together into the workflow of Section IV. The orchestrator decomposes incoming requirements into a task plan, generates and executes test cases for each task, invokes self-healing on flaky-failure candidates with a bounded retry budget and routes every result through the root-cause analysis agent before final reporting.

orchestrator.py

```python
# Orchestrator: Multi-Agent Test Execution Loop
def run_pipeline(requirements, config):
    plan = planner_agent.decompose(requirements)
    for task in plan.tasks:
        cases = gen_agent.generate(task)
        for case in cases:
            result = exec_agent.run(case, retries=config.max_retry)
            if result.failed and result.is_flaky_candidate():
                exec_agent.self_heal(case)
                result = exec_agent.run(case, retries=1)
            rca_agent.analyze(result)
    return reporter.build_dashboard(plan)
```

Figure 7: Orchestrator implementing the multi-agent execution loop with bounded self-healing retries.

D. Deployment considerations

The prototype was packaged as a set of containerized services triggered from a CI/CD pipeline on pull-request and nightly-schedule events. Prompts sent to the reasoning core were constrained to structured-output schemas to simplify downstream parsing and all agent-to-tool interactions were logged to support auditability, an important property given the non-deterministic nature of language-model outputs. Rate limiting and a bounded retry budget were applied at the orchestrator level to prevent runaway self-healing loops from delaying pipeline completion.

E. Technology stack summary

(Table 3) summarizes the concrete technologies used for each architectural component in the prototype, provided as a practical reference for teams evaluating a similar implementation.

Table 3: Technology stack by architectural component.

Component	Representative Technology / Tooling
Reasoning & planning core	LLM API (structured-output / function-calling mode)
UI / DOM perception	Playwright accessibility-tree snapshot, DOM parser
API / log observability	OpenAPI client, structured log ingestion
Test-case generation	Python service, Gherkin / framework-native script emitter
Test execution	Playwright, Selenium, Appium adapters
Self-healing locator	Multi-strategy resolver with vision-grounded fallback
Knowledge base / memory	Vector store for semantic retrieval, relational defect store
CI/CD orchestration	GitHub Actions / Jenkins, containerized runners
Reporting dashboard	Web dashboard consuming aggregated run metadata

6. Pilot Evaluation

To assess feasibility, the prototype was deployed against two representative targets: a customer-facing web application with an existing suite of 220 scripted UI test cases and a set of 60 REST endpoints supporting an internal order-management service. The evaluation was exploratory and qualitative rather than a controlled experiment and is intended to illustrate the behavior of the architecture rather than to establish statistically generalizable performance claims.

A. Test authoring

For 35 newly added user stories over a four-week pilot window, the generation agent produced an initial draft test case for each story; engineers reported that the majority of drafts required only minor edits - primarily assertion tightening - rather than being discarded, suggesting that requirement-grounded generation can materially reduce first-draft authoring effort relative to writing test cases from scratch.

B. Self-healing effectiveness

Across the pilot window, 46 previously passing UI test cases failed due to locator drift following unrelated front-end changes. The self-healing agent successfully recovered 39 of these cases without human intervention by falling back to vision-grounded matching, persisting the corrected locator for subsequent runs. The remaining seven cases involved structural navigation changes that the agent correctly flagged as requiring human review rather than attempting an unsafe automatic repair, illustrating the value of a bounded-autonomy design that defers ambiguous cases rather than silently masking a potential regression.

C. Root-cause triage

The triage agent classified 128 test failures observed during the pilot as regression, environment or flaky. Manual review by the engineering team concurred with the agent's classification for the large majority of cases, with residual disagreement concentrated in intermittent network-timing failures that are inherently ambiguous even for human reviewers, underscoring that automated triage should be treated as a decision-support aid rather than a fully autonomous gate in safety-critical contexts.

D. Summary of quantitative results

(Table 4) consolidates the pilot observations reported in Sections VI-A through VI-C into a single summary. Figures are drawn from the four-week pilot window described above and should be read as indicative of prototype behavior on the two target systems rather than as generalizable benchmarks.

Table 4: Summary of pilot observations over a four-week window.

Metric	Observed Value	Notes
New user stories drafted by generation agent	35 / 35	Majority required only minor assertion edits
UI tests affected by locator drift	46	Following unrelated front-end changes
Locator failures auto-recovered	39 (85%)	Vision-grounded fallback, no human intervention
Failures deferred to human review	7 (15%)	Structural navigation changes, flagged not repaired
Total failures triaged by RCA agent	128	Classified as regression, environment or flaky
Existing scripted UI test corpus	220	Baseline suite on the web application target
REST endpoints under test	60	Internal order-management service

E. Discussion of results

These observations are consistent with the architectural premise that separating perception, reasoning, execution and memory allows targeted automation of specific pain points - locator fragility and first-draft authoring effort in particular - without requiring the reasoning core to make unreviewed decisions about ambiguous or safety-relevant outcomes.

7. Challenges and Open Problems

Several challenges remain open for autonomous agentic testing systems. The oracle problem, long recognized by Barr et al., is not eliminated by language-model reasoning; an agent can generate plausible-looking assertions that do not correspond to genuine business intent, particularly when requirements are ambiguous or incomplete. Non-determinism in language-model outputs complicates reproducibility, since two runs of

the same generation agent may produce syntactically different but semantically equivalent test cases, which stresses version-control and audit practices designed around static scripted tests.

Explainability and trust present a further challenge: engineers must be able to understand why an agent classified a failure as flaky or chose a particular self-healing repair before granting the system broader autonomy, particularly in regulated domains. Bounded autonomy - deferring ambiguous decisions to human reviewers, as observed in the self-healing results of Section VI - is a practical mitigation but reduces the degree of automation achievable. Finally, the computational and monetary cost of invoking large language models for every generation and reasoning step motivates further work on selective invocation, caching of retrieved precedent and smaller specialized models for narrow sub-tasks such as locator classification.

8. Conclusion and Future Work

This paper presented a layered architecture and end-to-end workflow for autonomous, agentic software testing systems that combine large-language-model reasoning with specialized execution agents for test generation, self-healing execution and root-cause triage, coordinated through a continuously updated memory. A pilot deployment against a web application and a set of REST endpoints suggests the approach can reduce first-draft test-authoring effort and automatically recover a substantial share of locator-drift failures while deferring ambiguous cases to human review. Future work includes conducting controlled comparative studies against purely scripted and purely record-and-replay baselines, developing quantitative metrics for oracle quality and self-healing precision, exploring smaller fine-tuned models for latency- and cost-sensitive sub-tasks and extending the architecture to mobile and embedded testing contexts where perception of application state is substantially more constrained.

9. References

1. Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need. Advances in Neural Information Processing Systems (NeurIPS), 2017.
2. Devlin J, Chang MW, Lee K, et al. BERT: Pre-training of deep bidirectional transformers for language understanding. NAACL-HLT, 2019.
3. Brown T, Mann B, Ryder N, et al. Language models are few-shot learners in Advances in Neural Information Processing Systems (NeurIPS), 2020.
4. Wei J, Wang X, Schuurmans D, et al. Chain-of-thought prompting elicits reasoning in large language models in Advances in Neural Information Processing Systems (NeurIPS), 2022.
5. Yao S, Zhao J, Yu D, et al. ReAct: Synergizing reasoning and acting in language models. Int Conf on Learning Representations (ICLR), 2023.
6. Schick T, Dwivedi-Yu J, Dessì R, et al. Toolformer: Language models can teach themselves to use tools, 2023.
7. Park JS, O'Brien J, Cai CJ, et al. Generative agents: Interactive simulacra of human behavior. ACM Symp. on User Interface Software and Technology (UIST), 2023.
8. OpenAI. GPT-4 technical report, 2023.
9. Myers GJ, Sandler C, Badgett T. The Art of Software Testing, 3rd ed. Hoboken, NJ, USA: Wiley, 2011.
10. Barr ET, Harman M, McMinn P, et al. The oracle problem in software testing: A survey. IEEE Trans. Software Engineering, 2015;41: 507-525.
11. Zhang JM, Harman M, Ma L, et al. Machine learning testing: Survey, landscapes and horizons. IEEE Trans Software Engineering, 2022;48: 1-36.
12. Durieux T, Madeiral F, Martinez M, et al. Empirical review of Java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts. ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (ESEC/FSE), 2019.
13. Itkonen J, Mäntylä MV. Are test cases needed? Replicated comparison between exploratory and test-case-based software testing. Empirical Software Engineering, 2014;19: 303-342.
14. Wooldridge M, Jennings NR. Intelligent agents: Theory and practice. The Knowledge Engineering Review, 1995;10: 115-152.
15. Russell S, Norvig P. Artificial Intelligence: A Modern Approach, 4th ed. Hoboken, NJ, USA: Pearson, 2020.
16. International Software Testing Qualifications Board (ISTQB), Certified Tester Foundation Level Syllabus, 2018.